Appendix C

Abstract:

This paper presents a proposal for a new BIOS service, which will be known as the **BIOS32 Service Directory**. The new service will provide information about those services in the BIOS that are designed for callers running in a 32-bit code segment. (The BIOS32 Service Directory will itself be a 32-bit BIOS service.) The expected clients of this service are 32-bit operating systems and 32-bit device drivers. The expected providers of this service are BIOS vendors that implement one or more 32-bit BIOS services.

1 Introduction

The BIOS32 Service Directory proposal came into being during the attempts to establish a 32-bit code interface for the Peripheral Component Interconnect (PCI) standard. A major problem that needed solving was: How does a 32-bit caller determine the existence of a 32-bit BIOS service? The drawbacks of a functional interface (i.e., an entry point that control is transferred to) are clear in the case where the functional interface is non-extant. A recoverable method would seem to entail a search for a signature as proof that a functional interface does exist. In fact, one of the first BIOS 32-bit services, EISA, uses an existence signature at a fixed location in the 0F000h segment. However, as more 32-bit BIOS services are required and come into existence, it is obvious that providing a signature (and any associated information, such as entry points, segment requirements, etc.) for each is a costly usage of a scarce memory resource. Hence, the idea behind the

The BIOS32 Service Directory uses a single signature to indicate the existence of a generic 32-bit service that returns information on all specific 32-bit services

A further justification for implementing a general solution versus continuing to solve the problem on a case-by-case basis is seen in the obvious benefits a standard provides to the industry. Code reusability, modular ("plug-in"-able) implementation of new 32-bit BIOS interfaces, thorough specification of calling environment requirements, etc., are a few benefits which come to mind.

The BIOS32 Service Directory has two components: the Header and the Calling Interface. The Header is a static data structure that includes the signature and which is located in a well-known memory range. The existence of a valid Header implies the existence of the Calling Interface and, in fact, describes its entry point. The Calling Interface is a body of code and data which exists in a separate memory space apart from the Header and any particular 32-bit BIOS services. It provides a functional interface through which a caller receives information about a particular 32-bit BIOS service. (Section 0 describes a hypothetical memory map of the Header, Calling Interface, and the "XYZ" 32-bit BIOS service.)

The design of the Calling Interface is extensible in two dimensions. First, it is a function-based interface - future revisions of the service can incorporate new features, as they become necessary. Second, specific 32-bit BIOS services will be represented by a 4-byte Component ID. This will enable both OS callers and BIOS32 Service Directory implementors alike to easily add new 32-bit BIOS services as they become available.

Interface to the BIOS32 Service Directory, and a summary with example.

2 The BIOS32 Service Directory Header

A BIOS which implements the BIOS32 Service Directory must embed a specific, contiguous 16-byte pattern somewhere in the physical address range 0E0000h - 0FFFFFh. The pattern must be paragraph aligned (i.e., it must start on a 16-byte boundary). This pattern is known as the BIOS32 Service Directory Header.

The Header is comprised of six distinct fields. The following table describes each field.

Offset	Size	Description
0	4 bytes	The ASCII signature "_32_". This string is packed left to right: offset 0 is '_' (underscore), offset 1 is '3', offset 2 is '2', offset 3 is another '_'.
4	4 bytes	The entry point for the BIOS32 Service Directory Calling Interface. This is a 32-bit linear (i.e., not segment:offset) physical address.
8	1 byte	The revision level of the BIOS32 Service Directory Header and Calling Interface. The current revision is 0h.
9	1 byte	The length of the BIOS32 Service Directory Header. This is measured in units of paragraphs (16 bytes). The current Header has a length of 1h.
10	1 byte	The BIOS32 Service Directory Header checksum. This is a value which makes the cumulative ADD value of all bytes in the Header equal to 0h.
11	5 bytes	This field is reserved and should be set to 0h.

TABLE 1: The BIOS32 Service Directory Header

Clients of the BIOS32 Service Directory should first determine its existence by locating the Header. This is done by scanning 0E0000h to 0FFFF0h in paragraph increments and looking for a signature match ("_32_") in the first 4 bytes of each paragraph. When, and if, the signature is detected the client should perform a checksum of all bytes in the Header. (The Header length, in paragraphs, is found at offset 9h.) All bytes in the Header should ADD together with a result of 0h. If the checksum is valid then the 32-bit entry point field can be used as the address for the BIOS32 Service Directory Calling Interface. If the Header is not found then the BIOS32 Service Directory does not exist on the platform.

3 The BIOS32 Service Directory Calling Interface

If the BIOS32 Service Directory existence has been determined (via the Header test, above) then the 32-bit physical address found in the Header can be used as the entry point to the Calling Interface. Clients should CALL FAR to this address. The client's calling environment has the following requirements.

3.1 Code Segment

The CS code segment selector must be set up with a segment descriptor with the following values:

- The base address must be less than or equal to the (4kb) page address of the page that contains the entry point. For example, if the entry point is 0FFF81234h then the base address must be less than or equal to 0FFF81000h.
- The limit must be such that the base address plus the limit generate an address that is greater than or equal to the last address on the (4kb) page which follows the page containing the entry point. For example, if the entry point is 0FFF81234h then the base address plus the limit must be greater than or equal to 0FFF82FFFh.
- Simply stated, the base address and the limit must "encompass" both the page that contains the entry point and the following page.
- The segment type must be 100b (code, execute only) or 101b (code, execute/read). However, the implementors of the Service Directory cannot assume read access to the CS code segment.
- The system bit must be 1 (nonsystem segment).
- It is recommended that the Descriptor Privilege Level (DPL) be 0. (The CS descriptor DPL becomes the Current Privilege Level, or CPL). If the CPL is not 0, then the OS must provide trapping and virtualization services for ring 0 privileged instructions (such as those that access CRx). Note also the dependency of this field on the IOPL field in EFLAGS (see Section 0).
- The Default Size bit must be 1 (32 bits).

The BIOS32 Service Directory entry point, and its associated code and data, may be located anywhere within the 4Gb physical address space. However, it is guaranteed to be physically contiguous (i.e., it will be delivered in ROM or FLASH space) and to fit within two pages (i.e., it will not span three pages).

3.2 Data Segments

The DS data segment selector must be set up with a segment descriptor with the following values:

- The base address must be equal to the CS base address.
- The limit must be greater than or equal to the CS limit.
- The segment type must be 000b (data, read only) or 001 (data, read/write). However, the implementors of the Service Directory cannot assume write access to the DS data segment.
- The system bit must be 1 (nonsystem segment).
- The Descriptor Privilege Level (DPL) must be greater than or equal to CPL (see the DPL field in Section 0).

There are no requirements concerning the ES, FS, and GS data segment selectors.

3.3 Stack Segment

The SS stack segment selector must be set up with a segment descriptor with the following values:

- The segment type must be 011b (data, read/write, expand-down) or 001b data, read/write, expand-up).
- The system bit must be 1 (nonsystem segment).
- The Descriptor Privilege Level (DPL) must be equal to the CPL (see the DPL field in Section 0).
- The Default Size bit must be 1 (32 bits).
- The Granularity bit must be 1 (4Kb).

Note that the above settings ensure a stack size of at least 4kb. It is the caller's responsibility to ensure that there is at least 1kb of unused stack available.

3.4 Paging

Paging may or may not be enabled. If paging is enabled, then the address space that is described by the CS and DS selectors must be linearly contiguous. That is, the original physical contiguity of the Calling Interface as found in ROM or FLASH must be preserved. (The Calling Interface code and data is written to be position-independent and EIP-relative).

3.5 IOPL

In order for the Calling Interface to execute I/O instructions, the I/O Privilege Level (IOPL) field in EFLAGS must be greater than or equal to the CPL (see the DPL field in Section 0).

The BIOS32 Service Directory Calling Interface is function-based and all parameters are passed in registers. If a register is not specified as an output parameter for a function, then it will be preserved. All flags are preserved. Function values are passed as input parameters in register BL. Return status is passed back in register AL. A return status of 00h indicates that the function was successful. A return status of 80h indicates that the requested function (in BL) is not supported. Other AL return values are defined by the individual functions. There is currently one BIOS32 Service Directory function defined. It is specified below.

3.6 The Component Existence Function (BL = 0h)

The Component Existence function returns information about whether a specific 32-bit BIOS service exists and, if it does, what memory space it occupies.

Input:

BL, 0h

EAX, Component ID

The Component ID is a 4-byte ASCII string which uniquely identifies the 32-bit BIOS service. The specifications for particular BIOS services define their own Component IDs. (It is important that those specifications define whether the Component ID string is packed left to right, or right to left.)

EBX, Reserved, set to 0h

Output:

If requested 32-bit service does not exist

AL = 81h

If requested 32-bit service does exist

AL = 0h

EBX = base address of 32-bit BIOS service

ECX = length of 32-bit BIOS service

EDX = offset (from EBX) of 32-bit BIOS service entry point

The meaning of the EBX, ECX, and EDX registers is dependent on the particular 32-bit BIOS service specification. That is, they may represent exact values for setting up segment selectors, minimal "encompassing" values, etc.

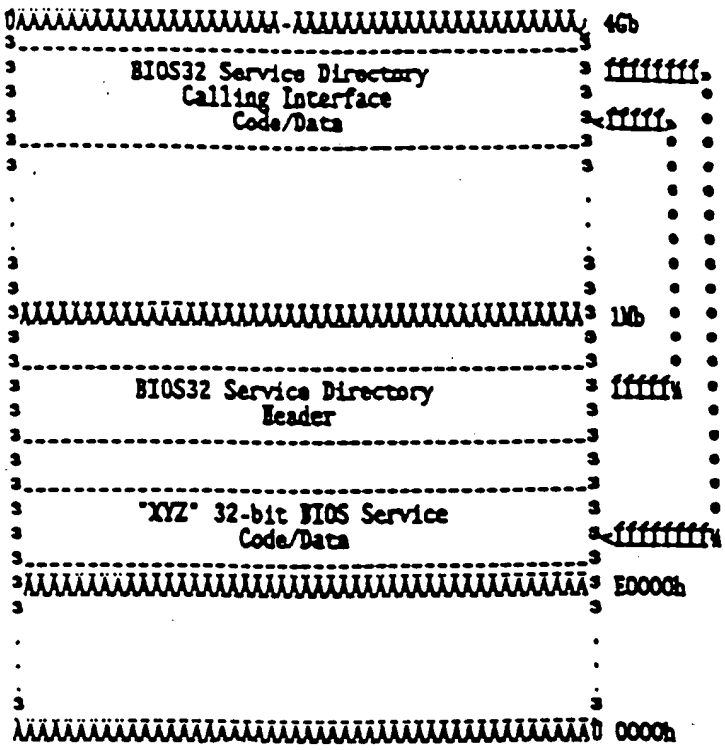
3.7 Future Functions

Future BIOS32 Service Directory functions may be defined in subsequent revisions of this document. The function parameter interface is not constrained to register passing and may employ input/output parameters on the stack. This is feasible due to the static definition of the stack (see Section 0).

4 Summary

This paper describes the BIOS32 Service Directory. It identifies two distinct memory components: the Header and the Calling Interface. The Header is a paragraph of signature data located in the memory range 0E0000h - 0FFFFFh. The Calling Interface is a body of code that occupies a physically contiguous section of memory anywhere in the 4Gb address space and is less than two pages (8kb) in length. The Header contains a pointer into the Calling Interface memory area.

The BIOS 32 Service Directory Calling Interface functions describe additional memory areas that contain code and data for specific 32-bit BIOS services. A memory map for a hypothetical Header, Calling Interface and the "XYZ" 32-bit BIOS service follows.



This paper provides for a standard method of accessing 32-bit BIOS services. The extensibility of the BIOS32 Service Directory Calling Interface may be used easily to advance a standard method as new 32-bit BIOS services are recognized and implemented.

```
#ifdef TESTEXEC

 //  test bios execute function.

 printf ("phadtest - test bios execution breakpoint\n");

 ioExec = (PIOC_EXEC1)systemBuffer;
 ioExec->biosFunction = biosShadowAreaBaseVA + biosServiceDirectoryVA;
 strncpy ((PUCHAR)(&ioExec->regEAX), BIOS_PM_SIGNATURE, 4);
 ioExec->regEBX = 0L;

 if (DeviceIoControl(hDriver,
      IOCTL_BIOS_EXEC,
      bufferPtr,
      256,
      bufferPtr,
      256,
      &actualXfr,
      NULL))  {
   printf ("IOCTL 2052 -BIOS Execute (%s) - success\n",
 BIOS_PM_SIGNATURE);
   printf ("# bytes returned: %Ld\n", actualXfr);

   printf ("Register contents:\n");
   printf ("AL  = %X\n",  (ioExec->regEAX ) & 0xff);
   printf ("EBX = %LX\n",  ioExec->regEBX );
   printf ("ECX = %LX\n",  ioExec->regECX );
   printf ("EDX = %LX\n",  ioExec->regEDX );

   }else{

    ioCode = GetLastError();
    printf ("IOCTL 2050 failed on %LX\n", ioCode);

    }

   #endif
```

Appendix D1

```
//-----------------------------------------------------------------------
   case IOCTL_BIOS_EXEC:

#ifdef DEBUGMZ
       MzIoKdPrint (4);
#endif

       //  Function 4 = Code 2052 is 'BIOS Execute'.
       //  The function executes code in the shadow area.

       ioExecBios = (PIOC_EXEC1) (Irp->AssociatedIrp.SystemBuffer);


       if (foundBios32 == FALSE) {          // if bios32 not found in.
        ioExecBios->runStat = FALSE;        // didn't run.
        return (IoCtlFinish (Irp, STATUS_SUCCESS, sizeof (IOC_EXEC1)) );
       }

       // check the range of the requested function.

       actualXfrd = rangeCheck (ioExecBios->biosFunction,    // function address.
                                1L,                          // only look at e.p.
                                MODE_SHADOW,                 // read/exec mode. BIOS.
                                lenBiosRom                   // length of rom space.
                               );

       // failed range check.

       if (actualXfrd == 0) {

           return (IoCtlFinish (Irp, STATUS_ACCESS_DENIED, sizeof (IOC_EXEC1)) );

       }


      // now execute the function.
      if ((ioExecBios->biosFunction)== ((ULONG)bios32ServiceEntryPoint+(ULONG)romMapPtr
          connectBios (ioExecBios, ioExecBios->biosFunction);
      else
          execBios (ioExecBios, ioExecBios->biosFunction);

      ioExecBios->runStat = 1;

          return ( IoCtlFinish(Irp, STATUS_SUCCESS, sizeof (IOC_EXEC1)) );
                    // don't forget to return exec struct..

   //-----------------------------------------------------------------------
```

Appendix D2

```
//  this function creates a register context and calls the
//  entry point.

void
execBios(
    PIOC_EXEC1      systembuffer,
    ULONG           entrypoint
    )
{

#ifdef BIOS_FAR_CALL

    void    (far *address32)(PIOC_EXEC1);

#else

    void    (*address32)(PIOC_EXEC1);

#endif


    *address32 = (void (*)(PIOC_EXEC1))entrypoint;

_asm {
    mov eax, [systembuffer]     // ptr to the systembuffer
    push    eax                 //
    push    cs                  // make it a FAR call
    call    [address32]         // call to the Serive Directory
}

//    (*address32)(systembuffer);           // this is service interface.

}
```

Appendix D3